Name: _____

Directions: **Work only on this sheet** (on both sides, if needed); do not turn in any supplementary sheets of paper. There is actually plenty of room for your answers, as long as you organize yourself BEFORE starting writing. In order to get full credit, **SHOW YOUR WORK**.

## An $\infty$ number of points will be deducted for illegibility.

**1.** (10) Consider the code

```
popl %ecx
popl %ecx
popl %ecx
```

(a) (10) Suppose we don't plan to use the popped values. Then fill in the blank in the instruction

```
addl __________, %esp
```

so that it will have an equivalent effect as the set of three pops.

(b) (10) How many bytes of memory are read or written (break it down accordingly) in the original code consisting of three pops? What about in the case of the add? Don't count instruction fetches.

**2.** Consider the material in Section 7.1 (titled "First Example") of our PLN on subroutines.

In parts (a) and (b), suppose in line 24 the assembly source code had accidentally been written as

```
call done
```

(a) (10) What would be the new machine language for this instruction?

(b) (10) Suppose we execute the program in GDB by placing a breakpoint at **done**, as usual. What will happen? (i) The program will produce a correct final sum in EBX, with an extra stop at the breakpoint. (ii) The program will produce an incorrect final sum in EBX, with an extra stop at the breakpoint. (iii) The program will produce a seg fault. (iv) The program will eventually go into an infinite loop.

Parts (c) and (d) concern the GDB session.

(c) (10) State the addresses at which the **.data** and **.text** segments begin.

(d) (10) Near the end of the session, I executed the GDB command **p sum**. What output would I have gotten if I had executed **p/x $esp**?

**3.** Consider Section 8.11 of our PLN on subroutines, titled "The Function main() IS..."

(a) (10) Suppose I were to run the program in GDB and am currently about to execute the CALL on line 22. Specify a GDB command I could execute which would print out the address of **main()**'s stack frame (in hex).

(b) (10) Suppose in the call to **printf()** we had asked to print out **argv[3]** instead of **argv[1]**. Which line in the assembly language would change, and what would it change to?

**4.** (10) Consider again the material in Section 7.1 of our PLN on subroutines. The following code, in which you will fill in the blanks, is to go between lines 25 and 26. It will replace the first value in the **x** array, currently 1, by a value to be read in from the keyboard.

```
pushl ________________________
pushl ________________________
pushl ________________________
pushl ________________________
pushl ________________________
call scanf
addl $8, %esp
popl ________________________
popl ________________________
popl ________________________
```

The overall program has had two other changes made to it:

```
fmt:   .string "%d"  # in the .data segment
...
# in place of _start
.globl main
main:
```

**5.** (10) Suppose we have an Intel-based PC with the keyboard as in Section 4 of our PLN unit on I/O, except that the keyboard ports have been attached to respond to MR and MW instead of IOR and IOW. Rewrite in C the assembly language code which reads one character.

**Solutions:**

**1.a** $12

**1.b** The original code reads 12 bytes, writes none; the new code does not access memory.

**2.a** The op code, according to the PLN, ps 0xe8. The distance from the next instruction after the call to **done** is 0x001b-0x0010 = 0x0b. Taking little-endianness into account, the instruction is then 0xe80b000000.

**2.b** The correct choice is (i). The CALL will cause a jump to **done**, after which execution will continue downward through the code, so that all of the code at and below **init** will be executed, as desired. When we hit the RET at offset 0x002e, that will cause a jump back to the saved return address, which the CALL had put on the stack as the address of the instruction immediately following the CALL. Again, that jump will be just what we want. "It all works out" (in this case, by accident).

**2.c** The queries to GDB show that just before the CALL was to be executed, c(EIP) = 0x804807f. The output

of **as -a** shows that the CALL was at offset 0x000b of the **.text** segment. So the latter must have started at 0x804807f-0x000b = 0x8048074.

The GDB session also shows that **x** was at location 0x80490a4. Since **x** was at the beginning of the **.data** segment, the latter must also start at 0x80490a4.

**2.d** Our first query to GDB about c(ESP) had been just before the CALL. At that time, the two arguments to **init()** had already been pushed onto the stack. The execution of the CALL and then the RET had no net effect on ESP, so when we reached the ADD which immediately followed the CALL, c(ESP) was still what it had been at our last query, 0xbfffcec8. The ADD then added 8 to that value, yielding 0xbfffced0. ESP never changed after that.

**3.a**

```
p/x $ebp
```

**3.b** The instruction

```
movl $12(%ebp), %eax
```

pointed EAX to the beginning of **argv**, i.e. **argv[0]**. The next line,

```
addl $4, %eax
```

moved EAX to point to **argv[1]**. If we want **argv[3]** instead, then the number added should be 12 instead of 4.

**4.** Of course, just before the CALL, we need to push the two arguments to **scanf()**, which are **$x** and **$fmt**. Note that the first of those should NOT be **x**, in contrast to the **printf()** case; that's because our call in the **scanf()** case would be **scanf("%",&x)**—note the ampersand.

But there's more. We have to worry about what **scanf()** might do to our register values. It says later in the notes that the C compiler's calling convention is as follows: The module being compiled, in this case **scanf()**, must save the values of the ESI, EDI and EBX registers that had been there at the time the calling module makes the call. In our case here, our calling module need not worry that its value in EBX may be lost. On the other hand, our calling module does use ECX, and the compiler does NOT guarantee that the value in that register will be preserved; so, we must save it on the stack before the call, and restore it after the call. We could also save and restore EDX for the same reason, in case we later add code which uses EDX.

In addition, in calling any C function, we know that there may be a return passed back in EAX. That is especially true for the C library functions, which generally give a result as a return value. So, in our calling module we must protect EAX too, by saving it before the call and restoring it afterward.

**5.**

```
char c1, // will store the character which is read
     c2, // will store a copy of the KSP Status
     *p;  // pointer to keyboard ports
...
p = 0x62;
while(1)  {
   c2 = *p;
   if (!(c2 & 0x10)) break;  // if KSP says key hit, break
}
c1 = *(p-2);  // read the character from the KDP
c2 |= 0x20;
*p = c2;  // flick ACK bit on
c2 &= 0xdf;
*p = c2;  // restore ACK bit to 0
```